Timed Automata Semantics for Analyzing Creol*

Mohammad Mahdi Jaghoori
LIACS, Leiden University, Leiden
CWI, Amsterdam, The Netherlands
jaghouri@cwi.nl

Tom Chothia
School of Computer Science
University of Birmingham, United Kingdom
t.chothia@cs.bham.ac.uk

We give a real-time semantics for the concurrent, object-oriented modeling language Creol, by mapping Creol processes to a network of timed automata. We can use our semantics to verify real time properties of Creol objects, in particular to see whether processes can be scheduled correctly and meet their end-to-end deadlines. Real-time Creol can be useful for analyzing, for instance, abstract models of multi-core embedded systems. We show how analysis can be done in UPPAAL.

1 Introduction

Parallel and distributed systems span a wide range of applications, from internet-based services to multi-core embedded systems. Concurrent objects, as in Creol [10], have dedicated processors and thus provide a natural way of modeling these systems. In a real-time setting, a major analysis problem is schedulability, that is, whether all tasks are accomplished within their deadlines. We enrich Creol with real-time and provide a timed-automata semantics for it where in particular schedulability can be analyzed; this semantics allows for explicit modeling of scheduling strategies. For practicality, we define this semantics in terms of a translation algorithm such that the generated automata can be handled by UPPAAL [15].

Each Creol method is mapped to exactly one automaton with the possibility of tracing from automata back to Creol code; the semantics of an object consists of a network of the automata for its methods together with a scheduler automaton (e.g., Earliest Deadline First, Fixed Priority Scheduling, etc.). We have implemented a program that translates Creol code to the corresponding timed automata, which can then be run and verified in UPPAAL. We validate the automated verification cycle by checking a coordinator class by Johnsen and Owe [10], augmented with real-time, for correctness as well as schedulability.

A distinctive feature of Creol is that it gives the programmer a high degree of control over the scheduling. At "processor release points", a method can choose to give up control of the processor and specify conditions on when it should be resumed. Previous work on Creol has not addressed particular scheduling policies; instead processes are scheduled in a completely non-deterministic manner. The real time extensions and schedulability analysis we present here can be seen as a complementary feature for the basic Creol language.

In Creol, object references are typed only by interfaces, allowing for a strongly typed language with support for features such as multiple inheritance and type-safe dynamic class upgrades [18]. We augment an interface with a notion of high-level behavioral specification given in a timed automaton; this is given by the modeler as the first step in modeling an object. A class can implement multiple interfaces. In this case, the timed automata for behavioral interfaces should be interleaved; intuitively, allowing more behavior than each automaton separately. The justification is that a class with multiple interfaces can

*This work is partly funded by the European IST-33826 STREP project CREDO on Modeling and Analysis of Evolutionary Structures for Distributed Services.

M.R. Mousavi and G. Salaün (Ed.): International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2010).
EPTCS 30, 2010, pp. 108–122, doi:10.4204/EPTCS.30.8

participate in multiple protocols independently. Creol supports multiple inheritance for both interfaces and classes. Interfaces form a subtype hierarchy, which is distinct from inheritance at the level of classes used merely for code reuse [11]. In a subtype hierarchy, the timed automata for the inherited behavioral interfaces should synchronize on similar actions; intuitively, allowing less behavior in the subtype. The justification is that a subtype should be a refinement of its supertypes [17].

We use a statement-based delay semantics for our notion of real-time in Creol, namely, we annotate each statement with best- and worst-case execution times, as it naturally fits the operational semantics of Creol and it is the normal trend in schedulability analysis [3, 6, 7, 12]. Also, all method calls are given a deadline, which specifies the time before which the callee *should* finish. By giving the annotations in comments, we allow the standard Creol interpreter to behave normally by ignoring the real-time information if necessary. One could specify cumulative delays for a group of statements (by assigning the default delay of zero to all statements except the last one). When a Creol model is schedulable, it gives us requirements on point-to-point execution times that should be fulfilled by the final implementation in programming language like C; this can also be tested by checking conformance [1].

Whenever a called method finishes, it sends a 'reply' back to the caller. By waiting for the reply, one can model synchronous method calls. Timely replies ensure end-to-end deadlines. Method invocations are labeled to help match replies with the originating invocations. Dynamic labels lead to infinite state models for non-terminating systems. To model this using finite state timed automata, we label method invocations statically; thus, replies to repeated invocations associated with the same label are not distinguished in our automata semantics. Furthermore, we abstract from other dynamic statements in Creol. Although in our semantics, we abstract from the dynamic behavior, i.e., dynamic object creation and reconfiguration, it is worth noting that these actions have no effect in individual object analysis.

In Section 2, we briefly introduce timed automata and UPPAAL models. Section 3 describes Creol and discusses how it can be extended with real time. In Section 4 we discuss how Creol with real time can be given a semantics using timed automata. We show how this semantics can be applied to analyze schedulability in Section 5. We conclude in Section 6.

Related Work In previous work [8, 9], we provided a high-level framework for modular schedulability analysis of purely asynchronous objects modeled as timed automata. This past work assumes methods run until completion without interruption, unlike the work we present here. In this paper we apply this automata theoretic framework to the concurrent object modeling language Creol [10] by defining a mapping from Creol processes to timed-automata; to accomodate Creol features, the scheduler model is also extensively extended.

We discussed a possible encoding from Creol to timed automata in a past paper [4], however this encoding was incomplete, both in the Creol features covered and because it discussed only translation of one method; it would generate many automata to encode each method. In the full, efficient encoding presented in this paper, we improve scalability by reducing the number of generated automata (i.e., release statements do not generate a new automaton now). Further, we improve label handling by making the scheduler responsible for this; we add features like multiple interfaces; blocking statements are now also handled by the scheduler; our improved scheduler allows for local synchronous calls, which may incur recursive calls. With this semantics, we can now verify Creol models for schedulability; examples of other properties we can check include deadlock, or timed reachability of any specific line of code.

Creol$_{\mathrm{RT}}$ [14] is a real-time extension of Creol along the lines of Hoare logic extended with real-time. In Creol$_{\mathrm{RT}}$, time can affect the functional behavior of the object, but we use only descriptive annotations. It is possible in Creol$_{\mathrm{RT}}$ to specify contradicting invariants, which is not the case in our simple delay model. By automatically generating timed automata, we can readily use UPPAAL for model checking and

schedulability analysis, however, to the best of our knowledge, Creol_{RT} has no automated tool support. Another important difference is that we can model and compare explicit scheduling strategies, and so say which strategies would and would not lead to missed deadlines.

With respect to schedulability analysis, a characteristic of our work is modularity. A behavioral interface models the most general input/output behavior allowed for an object and thus can be used as an abstraction of the environment. A behavioral interface can be viewed as a contract as in "design by contract" [16] or as a most general assumption in modular model checking [13] (based on assume-guarantee reasoning); schedulability is guaranteed if the real use of the object satisfies this assumption. In the literature, a model of the environment is usually the task generation scheme in a specific situation. For example, in TAXYS [6], different models of the environment can be used to check schedulability of the application in different situations. However, a behavioral interface in our analysis covers all allowable usages of the object, and is thus an over-approximation of all environments in which the object can be used. This adds to the modularity of our approach; every use of the object foreseen in the interface is verified to be schedulable. Finally, TAXYS deals with a programming language, whereas we work at modeling level. It is also the case with the works of [12] where they extract automata from code for schedulability analysis. As mentioned above, they deal with programming languages (like TAXYS) and timings are usually obtained by profiling the real system. Our work, on the contrary, is applied on a model before the implementation of the real system. Therefore, our main focus is on studying different scheduling policies and design decisions. These models can be used for automatic code generation or conformance checking with later or existing implementations.

2 Preliminaries

We give a brief introduction to Timed Automata, for a full description we refer the reader to previous papers on timed automata (e.g. [2]) and documentation for the UPPAAL tool [15].

Definition 1 (Timed Automata). Suppose $\mathcal{B}(C)$ is the set of all clock constraints on the set of clocks C. A timed automaton over actions Act and clocks C is a tuple $\langle L, l_0, \longrightarrow, I\rangle$ representing

- a finite set of locations L (including an initial location l_0);
- the set of edges $\longrightarrow \subseteq L \times \mathcal{B}(C) \times Act \times 2^C \times L$; and,
- a function $I : L \mapsto \mathcal{B}(C)$ assigning an invariant to each location.

An edge (l, g, a, r, l') implies that action 'a' may change the location l to l' by resetting the clocks in r, if the clock constraints in g (as well as the invariant of l') hold. A timed automaton is called *deterministic* if and only if for each $a \in Act$, if there are two edges (l, g, a, r, l') and (l, g', a, r', l'') from l labeled by the same action a then the guards g and g' are disjoint (i.e., $g \wedge g'$ is unsatisfiable). Since we use UPPAAL [15], we allow defining variables of type boolean and bounded integers. Variables can appear in guards and updates.

A system may be described as a *network* of communicating timed automata. In these automata, the action set is partitioned into input, output and internal actions. The behavior of the system is defined as the parallel composition of those automata $A_1 \parallel \cdots \parallel A_n$. Semantically, the system can delay if all automata can delay and can perform an action if one of the automata can perform an internal action or if two automata can synchronize on complementary actions (inputs and outputs are complementary). In a network of timed automata, variables can be defined locally for one automaton, globally (shared between all automata), or as parameters to the automata.

b : Boolean	in	$::=$	$\textbf{interface}\ n([n:n]^*_,)\ [\textbf{inherits}\ [n[(e^+_,)]^?]^+_,]^?\ \textbf{begin}\ [[\textbf{with}\ n]^?\ \textbf{op}\ n]^+\ \textbf{end}$
e : Expression	cl	$::=$	$\textbf{class}\ n([n:n]^*_,)\ \textbf{implements}\ [n[(e^+_,)]^?]^+_,\ \textbf{begin}\ [\textbf{var}\ Vdcl]^*\ [mtd]^+\ \textbf{end}$
g : Guard	$Vdcl$	$::=$	$n^+_, : [int \mid bool]$
m : Method	mtd	$::=$	$[\textbf{with}\ n]^?\ \textbf{op}\ n\ ==\ S$
n : Identifier	S	$::=$	$s \mid s;S$
s : Statement	s	$::=$	$n := e \mid !x.m() \mid t!x.m() \mid t? \mid \textbf{release} \mid \textbf{await}\ g$
t : Label			$\mid \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od} \mid \textbf{if}\ b\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{skip}$
x : Object	g	$::=$	$b \mid t? \mid \sim g \mid g \wedge g$

Figure 1: BNF grammar for Creol (adapted from [10]) where * and $^+$ show repetition for at least 0 and 1 times, respectively; $^?$ denotes an optional element; and, a subscript $_,$ implies a comma-separated list.

A location can be marked *urgent* in an automaton to indicate that the automaton cannot spend any time in that location. This is equivalent to resetting a fresh clock x in all of its incoming edges and adding an invariant $x \leq 0$ to the location. In a network of timed automata, the enabled transitions from an urgent location may be interleaved with the enabled transitions from other automata (while time is frozen). Like urgent locations, *committed* locations freeze time; furthermore, if any process is in a committed location, the next step must involve an edge from one of the committed locations.

Definition 2 (UPPAAL Model). An UPPAAL model consists of: (1) a set of timed automata templates (TAT); (2) global declarations; and, (3) system declarations.

An automata template in an UPPAAL model consists of a name, a set of arguments, local declarations and a timed automaton definition (as above); formally, $TAT = (tName, Args, local, Auto)$. Global and local declarations contain the definition of clocks and variables. The network of timed automata to be analyzed is defined in the system declarations by instantiating the timed automata templates.

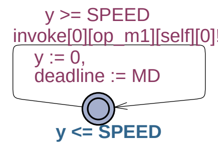
3 Creol with Real-Time

Creol [10] is an object oriented modeling language for distributed systems, where each object implicitly has a dedicated processor. A simplified syntax for Creol, for which we give a semantics is shown in Figure 1. We use, as running example, a coordinator class, taken from [10], concretized for three-way synchronization (while abstracting data away), shown in Figure 2.

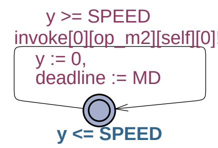
A Creol model is defined as objects typed by interfaces, which are in turn a set of method definitions; a method can define a cointerface using the `with` keyword to restrict the type of its caller to the given interface. A class implementing some interfaces realizes their methods possibly introducing private methods and local variables. With inheritance of interfaces, one can create a subtype hierarchy. We abstract from method parameters and dynamic object creation. However, classes and interfaces can have parameters. Instances of a class can communicate by objects given as class parameters, called the *known objects*. We can thus define the static topology of the system. The class behavior is defined in its methods, where a method is a sequence of statements separated by semicolon. For expressions, we assume the syntax that is accepted by UPPAAL. The `Coordinator` class implements three interfaces, each with one method, e.g., `with Any op m1` that defines method `m1` which can be called from objects of *any* type. The method `init` in an object is immediately executed upon object creation. The method `run` specifies active behavior of the object; it will have to compete with other tasks in the queue if any.

Methods can have processor release points which define interleaving points explicitly. When a process is executing, it is not interrupted until it finishes or reaches a release point. Release points can be

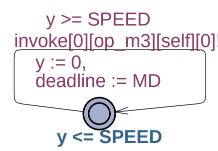
```
1  interface C1 begin with Any op m1 end
2  interface C2 begin with Any op m2 end
3  interface C3 begin with Any op m3 end
4  class Coordinator implements C1, C2, C3 begin
5    var s1,s2,s3,sync : bool
6    op init ==
7      s1 := false; // assume zero exec. time by
8      s2 := false; // default, resulting in a
9      s3 := false; // collective delay
10     sync := true /* specified here: @b2 @w4*/
11   op body ==
12     skip                          /*@b2 @w5*/
13   op run ==
14     await (s1 /\ s2 /\ s3);       /*@b1 @w2*/
15     b!body();  /* invoc. delay @b1 @w1 @d10*/
16     b?;        // force sync call
17     sync := false;
18     await (~s1 /\ ~s2 /\ ~s3);    /*@b1 @w2*/
19     sync := true;                 /*@b1 @w1*/
20     !run()                        /*   @d50*/
21   with Any
22   op m1 ==
23     await (sync /\ ~s1);
24     s1 := true;                   /*@b1 @w1*/
25     await ~sync;
26     s1 := false                   /*@b1 @w1*/
27   op m2 ==  ...like m1...
28   op m3 ==  ...like m1...
29 end
```



Behavioral specification for C1



Behavioral specification for C2



Behavioral specification for C3

Figure 2: A real-time coordinator in Creol with periodic task generation in the behavioral interfaces.

conditional, written as `await g` (e.g., line 14). If `g` is satisfied, the process keeps the control; otherwise, it releases the processor. A suspended process will be enabled when the guard evaluates to true. When the processor is free, an enabled process is nondeterministically selected and started. The `release` statement unconditionally releases the processor and the continuation of the method is an enabled process.

If a method invocation `p` is associated with a label `t`, written as `t!p()`, the sender can wait for a reply using the blocking statement `t?` or in a nonblocking way by including `t?` in a release point, e.g., as in `await t?`. A reply is sent back automatically when the called method finishes. Before the reply is available, executing `await t?` releases the processor whereas the blocking statement `t?` does not. While the processor is not released, the other processes in the object do not get a chance for execution; if `t?` is related to a self call and its reply is not yet available, it forces synchronous execution of the called method. For example, line 16 in Figure 2 forces synchronous execution of the method `body`.

Adding Real-Time The modeler should specify for every statement how long it takes to execute. The directives `@b` and `@w` are used for specifying the best-case and worst-case execution times for each statement. We assume zero execution time for statements with no annotations; there must be, however, non-zero execution time in each process before the processor is released. Furthermore, every method

call, including self calls, must be associated with a deadline using `@d` directive. This deadline specifies the relative time before which the corresponding method should be scheduled and executed. Since we do not have message transmission delays, the deadline expresses the time until a reply is received. Thus, it corresponds to an *end-to-end* deadline. A worst-case execution time delay for a blocking statement `t?` is ignored. The delay associated to release statements specifies only the time for invoking the command and not the waiting time afterwards.

Creol interfaces are enriched with behavioral interfaces given in timed automata, which specify the abstract behavior of an object. This interface consists of the messages the object may receive and send and provides an overview of the object behavior in a single automaton. It should also contain the reply signals the object may receive. A behavioral interface abstracts from specific method implementations, the queue in the object and the scheduling strategy. Figure 2 shows the behavioral interfaces for the coordinator example that assume periodic arrival of events. To formally define behavioral interfaces in UPPAAL, we assume two finite global sets $\mathcal{M}$ for method names and $\mathcal{T}$ for labels; and, two UPPAAL channels invoke and reply (cf. modeling schedulers in the next section).

Definition 3 (Behavioral interface). A behavioral interface B, with known objects $\mathcal{K}$, providing a set of method names $M_B \subseteq \mathcal{M}$ is formally defined as a deterministic timed automaton over alphabet Act^B such that Act^B is partitioned into three sets of actions (assume $t \in \mathcal{T}$ and $k \in \mathcal{K}$):

- object inputs sent by the environment: $Act_I^B = \{\mathrm{invoke}[0][m][\mathrm{self}][k]! \,|\, m \in M_B\}$
- object outputs received by the environment: $Act_O^B = \{\mathrm{invoke}[t][m][k][\mathrm{self}]? \,|\, m \in \mathcal{M} \wedge m \notin M_B\}$
- replies to the object outputs: $Act_r^B = \{\mathrm{reply}[t][\mathrm{self}]!\}$

Transitions specifying an input to the object must update the variable deadline with an integer specifying the deadline for that input. If B has no known objects then $\mathcal{K} = \{0\}$ by default.

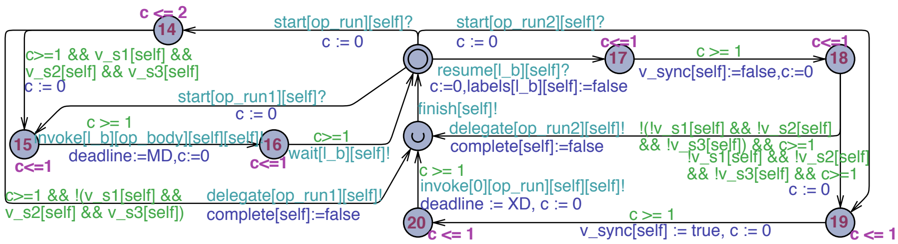
4 Timed Automata Semantics For Creol

The semantics of a Creol class consists of the automata for its methods. When instantiating a class, it should also be associated with a scheduler automaton. In this section, we explain the algorithm for automatically deriving automata from Creol code. This semantics maps each method to one automaton. For practicality, we define this semantics in terms of a translation algorithm that generates an UPPAAL model containing the automata for methods together with other necessary global declarations.

Definition 4 (Class). The semantics of a class R implementing a set of behavioral interfaces B^* is defined as follows. Recall that each $B \in B^*$ has a set of method names M_B. R is a set $\{(m_1, E_1, A_1), \ldots, (m_n, E_n, A_n)\}$ of tasks, where

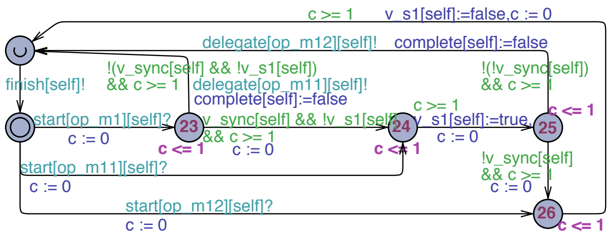
- $M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of task names such that $M_B \subseteq M_R$ for every $B \in B^*$. M_R includes the sub-tasks created at release points, as well as the methods; however, there is exactly one automaton for each method.
- for all i, $1 \leq i \leq n$, A_i is a timed automaton representing the method containing the task m_i.
- for all i, $1 \leq i \leq n$, E_i is the enabling condition for m_i.

We assume that the given Creol models are correctly typed and annotated with timing information. We use the same syntax for expressions and assignments in Creol, as is used by UPPAAL. This allows for a more direct translation. For the sake of simplicity, we abstract from parameter passing, however, it can be modeled in UPPAAL by extending the queue to hold the parameters (cf. Section 4.1). Figure 3



Task	Enabling Condition
run	true
run1	s1 ∧ s2 ∧ s3
run2	∼s1 ∧ ∼s2 ∧ ∼s3

(a) The automaton for method `run` and the enabling conditions for its subtasks.



Task	Enabling Condition
m1	true
m11	sync ∧ ∼s1
m12	∼sync

(b) The automaton for method `m1` and the enabling conditions for its subtasks.

Figure 3: The labels on automata location refer to line numbers in Figure 2 (cf. function *Loc* in Table 1).

shows the automata generated for the methods run and m1 from Figure 2. The automata for m2 and m3 are similar. As can be seen in these automata, the method and variable names are prefixed in order to avoid name clashes with UPPAAL keywords. As explained in Section 4.2, function *Loc* will link each automata location to its corresponding part of the original code; this is shown in Figure 3 as line numbers refering to the original Creol code in Figure 2.

Methods may release the processor before their completion, as in the 'await' statement in line 14 in Figure 2. In these cases, the rest of the method is modeled as a sub-task, e.g., the transition from location 14 to `U` in Figure 3 generates a sub-task op_run1 if the guard associated to await does not hold; the processor is released by the subsequent transition with the action finish going back to the initial location. This sub-task inherits the remaining deadline of the original task; this is done by the scheduler when handling the delegate channel (see next subsection). The enabling condition of the sub-tasks are equivalent to the guard used in the corresponding release point (see tables in Figure 3). The sub-task op_run1 can be triggered with the start transition entering location 15.

In standard Creol, different invocations of a method call are associated with different values of the label. For instance executing the statement t!p() twice results in two instances of the label t. Dynamic labels give rise to an infinite state space for non-terminating reactive systems. To be able to use model checking, we treat every label as a static tag. Therefore, different invocations of a method call with the same label are not distinguished in our framework. Alternatively, one could associate replies to message names, but this is too restrictive. By associating replies to labels, we can still distinguish the same message sent from different methods with different labels.

We handle labels with a global boolean array labels, and assuming that label names are unique in each class, we can define constants for each label such that labels[t][self] uniquely identifies label *t*. When the condition in a release point includes t?, i.e., waiting until the reply to the call with label *t* is available, we replace *t* with labels[t][self] which is set to true by the scheduler when called method finishes. For outgoing messages, the behavioral interface should capture when a reply is expected (cf. Definition 3).

Blocking statements use the wait channel to transfer the control to the scheduler, which then can

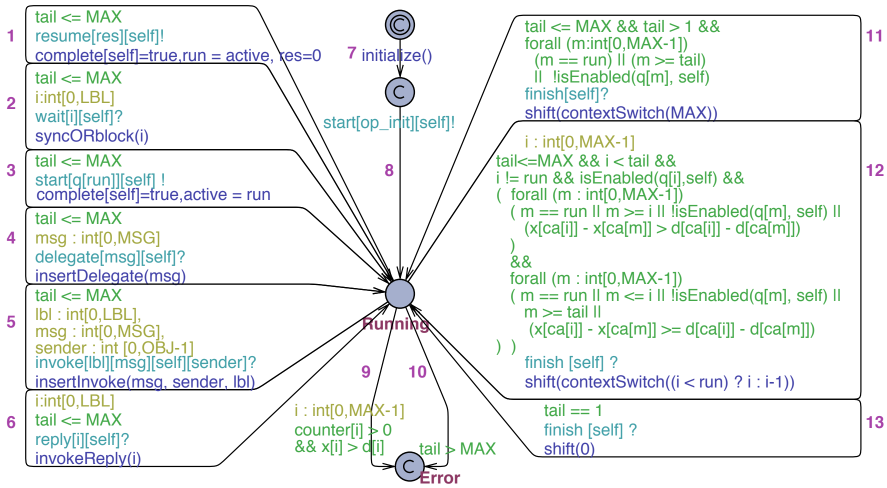


Figure 4: An Earliest-Deadline-First scheduler modeled in UPPAAL (details explained in the text).

check whether the label is associated to a local call; thus it can decide whether to block the object or do a synchronous call. To allow recursive synchronous calls, the wait transition goes back to the initial state making the method reentrant, e.g., the transition from location 16 modeling the statement b?.

Another complication in translation is how to map a possibly infinite state Creol model to finite state automata. We do this by abstracting away some information. One automatic way of abstracting is as follows: variables from a finite domain can be mapped to themselves but conditions on potentially infinite variables are mapped to true, we perform this with the function *Abs* in Figure 5. We do it semi-automatically, i.e., the user states which variables are abstracted away. In other words, we over-approximate the behavior of the Creol model. A more advanced abstraction would map potentially infinite variables to finite domains in order to narrow the over-approximation.

4.1 Modeling schedulers

A scheduler manages the buffering and execution of the tasks. A typical *scheduler automaton* is given in Figure 4. A transition numbered α in this figure is referred to as t_α in the text. This model is generic except for the strategy, explained below. The keyword self holds the current object identifier. Local declarations including function definitions are not given due to space limitations.

Queue The queue has the size MAX. A scheduler automaton uses multiple arrays of size MAX to implement the queue and its clocks. The message in q[i] is received from object s[i]; and, ca[i] points to a clock in array x that records how long the message has been in the queue: this message should be processed before the clock x[ca[i]] reaches its deadline value d[ca[i]]. A clock x[i] may be assigned to more than one task, the count of which is stored in counter[i]. We have shown in [8] that schedulable objects need bounded queues, i.e., with a proper MAX value, queue overflow implies nonschedulability. An Error location is reachable when a queue overflow occurs (t_{10}) or a task in the queue explicitly misses its deadline (t_9). Support for parameters can be added with an array p such that p[i][j] holds the jth parameter of q[i].

Channels The start channel is for starting execution of a task (t_3 and t_8), which later gives up the processor with a signal on finish channel (t_{11}, t_{12} and t_{13}). The invoke channel is for sending/receiving messages

(async call); t_5 generates a new task with the deadline given in the variable deadline by putting the message name, sender and the associated label in the queue. The delegate channel generates a sub-task (t_4), which inherits the deadline of the parent task. A reply to an async call is reported on reply channel (t_6). The blocking statement t? passes control to the scheduler using wait channel (t_2); this allows the scheduler to perform a synchronous call when needed. A blocked task is resumed using resume channel (t_1). To avoid context-switch delays, start and resume defined urgent.

Start-up An object is initialized by putting 'init' and 'run' methods in its queue (t_7), immediately followed by executing the 'init' method (t_8). During 'init', the object may receive other messages which are allowed to compete with the 'run' method for execution.

Context-switch Tasks can have enabling conditions, which may include the availability of a reply, but does not depend on clock values. Therefore, we can define in UPPAAL a boolean function isEnabled to evaluate the enabling condition for each method when needed. Whenever a task finishes, the scheduler selects another *enabled* task, based on its strategy (t_{12}), sets run to point to this task, and executes it (t_3). There are two special cases: (1) the last task in the queue has just finished (t_{13}); since run is zero and q[0] gets EMPTY after shift, the processor becomes idle until a new task arrives (t_5). (2) all remaining tasks in the queue are disabled (t_{11}); run is set to MAX to block the processor. In this case, the object may be enabled either by receiving a new task (t_5) or receiving a reply signal (t_6). The functions insertInvoke and invokeReply take care of these cases.

Labels The array labels associates a boolean to each label. Upon completion of a method, the label used for its invocation is set to true in the shift function (t_{11}, t_{12}, t_{13}). Since these transitions fire also at processor release points, the function shift uses the variable complete. This variable is true by default and will be reset to false at release points.

Synchronous call Executing the blocking t? statement (wait channel) on a remote call blocks the object until a reply is received. However, if t is associated to a local call, the scheduler will start the called method (modeling synchronous function call); upon termination of the called method, the scheduler will resume the blocked process (resume channel). In order to allow nested synchronous invocations, each message in the queue stores a pointer to the caller method. As explained in the next section, the blocked method goes back to its initial location allowing recursive synchronous function calls. Note that this recursion is bounded with the queue length.

Scheduling strategy The selection strategy is specified as a guard on t_{12}. Parts of this guard ensure that we consider only non-empty queue elements (i < tail) containing an enabled task (by calling isEnabled) different from the currently running one (run). Figure 4 compares the remaining deadline of task i, obtained by d[ca[i]] – x[ca[i]], with other tasks in the queue; task i is selected when its deadline is strictly less than that of task m for m>=i and less than or equal for m<=i. By replacing deadlines with a priori defined task priorities, we can model fixed-priority-scheduling. By fixing run to zero, we obtain first-come-first-served strategy. Nevertheless, the model is not restricted to these strategies.

4.2 A Formal Encoding from Creol Syntax to Timed Automata

We define this semantics in terms of a translation algorithm. The input to this algorithm is a Creol model consisting of class and interface specifications. The output is one UPPAAL model for each class; this UPPAAL model consists of only a set of timed automata templates (*TAT*), one for each method, and the global declarations (*Dec*). The system declarations is not generated automatically, because it depends, among others, on the choice of scheduling strategy (cf. Section 5). We formally define the

Function	Input	Output
$\underline{[\![.]\!]} : M \to A$	A method	A timed automaton
$\underline{[\![.]\!]} : S \times L \times L \to T \times L \times I \times 2^{N \times G}$	A Creol statement, two automata locations	Part of a timed automaton (transitions, locations, invariants), set of enabling conditions
$Loc : L \to S$	A location in a method automaton	A Creol statement in the method body
$LabelReset : G \to 2^U$	A guard	UPPAAL update statements
$Abs : E \to E$	Creol expression	UPPAAL expression
$labels : * \to L$	overloaded	label names
$mthds : * \to N$	overloaded	method names

Table 1: Some functions used for translation

translation for a class: $[\![\textbf{class } C\ (a^*)\ \textbf{implements } i^*\ \textbf{begin } v^*\ m^*\ \textbf{end}]\!] = (Dec, TAT)$, where $TAT = \{(Beh(I), C\_I, Args, Local(I)) \mid I \in i^*\} \cup \{(\underline{[\![\textbf{op } N == S]\!]}, C\_N, Args, \{\}) | \text{“}\textbf{op } N == S\text{”} \in m^*\}$. Functions $Beh(I)$ and $Local(I)$ return the user defined automata and their local declarations for the interface with the name I. All automata templates have as arguments: $Args = \{$`const int` arg`;` $\mid arg : Type \in a^*\} \cup \{$`const int self;`$\}$. First we define automata templates for methods and later Dec in this section.

Automata Templates The timed automata for each method is obtained using the function $\underline{[\![.]\!]}$ defined in Figure 5. For the translation of each method, the locations l_0 and u refer to the unique locations representing the *initial* and *final* location of the method automaton, respectively. The final location u is urgent and is connected to l_0 to allow multiple incarnations of the method execution.

The partial function $Loc : L \to S$ assigns to the locations of a method automaton, their corresponding statements in the method body. This function is not defined for the initial and the urgent final locations in the generated method automata. With each automaton corresponding to exactly one method, one can use this function to trace from automata back to methods. In Figure 3, this function is depicted as labels on locations, referring to line numbers in Figure 2.

The function $\underline{[\![S]\!]}_{a,e} = (T, L, I, E)$ translates the statements S to a set of transitions T on locations L and with location invariants I. Processing S should start from the location a and finish at e (obviously L will contain a and e). Additionally, E will correspond to a set of pairs (n, en), where n is the name of a sub-task (generated by a release point) with the enabling condition en (e.g., see tables in Figure 3).

Given a method **op** $N\ ==\ S$, the corresponding timed automaton is computed as shown in Figure 5 by computing $\underline{[\![\textbf{op } N\ ==\ S]\!]}$, which uses the overloaded function $\underline{[\![.]\!]}$ to compute the automaton transitions. A cointerface given using **with** keyword is ignored. The intuition behind the translation function is given on the right hand side of Figure 5 using a notation similar to graph transformation rules.

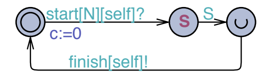
The translation uses some helper functions. As explained in the scheduler, there is a global boolean array $labels$. For every label t that is checked in guard g, $LabelReset(g)$ resets $labels[t]$ to false. $Abs(ex)$ is the abstract of the expression or assignment ex with proper renaming of the variables; the abstract must map the expressions to a finite domain, mapping everything to $true$ is correct but not optimal. If the domains of the variable are finite we can leave them unchanged, however if they are possibly infinite then we must approximate them. The user of our translation can decide the exact approximation, as it will need to balance between state-space size and accuracy. A possible definition of these functions are:

$LabelReset(g \wedge g') = LabelReset(g) \cup LabelReset(g')$
$LabelReset(!g) = LabelReset(g)$
$LabelReset(l?) =$ `labels[l][self]=false;`
$LabelReset(b) = \{\}$

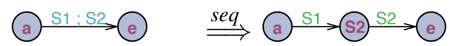
$Abs(ex) = ex\{$`labels[l][self]/l?`$\}\{$`v[self]/v`$\}$ for variables v and labels l.

Next, we explain two most complex rules with the example in Figure 3.

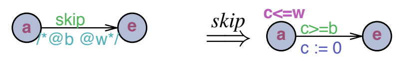
$[\![\textbf{with } N' \textbf{ op } N \ == \ S]\!] = [\![\textbf{op } N \ == \ S]\!] = (L \cup \{l_0, a, u\}, l_0, T \cup T_1, I')$ where $[\![S]\!]_{a,u} = (T,L,I,E)$; and, $I' = \{(a, c \geq w) \mid (a,w) \in I\}$; and,
$T_1 = \{l_0 \xrightarrow[c:=0]{start[N][self]?} a, u \xrightarrow{finish[self]!} l_0\}$; and,
New locations: l_0 (initial), u (urgent) and a such that $Loc(a) = S$



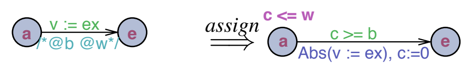
$[\![S_1;S_2]\!]_{a,e} = [\![S_1]\!]_{a,l} \uplus [\![S_2]\!]_{l,e}$
New location: l such that $Loc(l) = S_2$



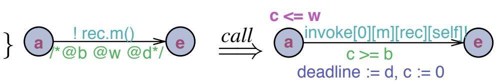
$[\![skip\ /*@b\ @w*/]\!]_{a,e} = \{a \xrightarrow[c \geq b\ ;\ c:=0]{} e\}, \{(a,w)\}$



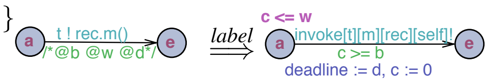
$[\![v := ex\ /*@b\ @w*/]\!]_{a,e} = \{a \xrightarrow[c \geq b\ ;\ c:=0, Abs(v:=ex)]{} e\}, \{(a,w)\}$



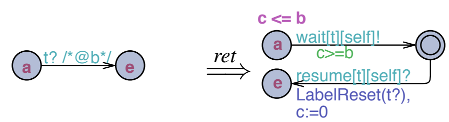
$[\![!rec.m()\ /*@b\ @w\ @d*/]\!]_{a,e} = \{a \xrightarrow[c \geq b\ ;\ c:=0, deadline:=d]{invoke[0][m][rec][self]!} e\}, \{(a,w)\}$



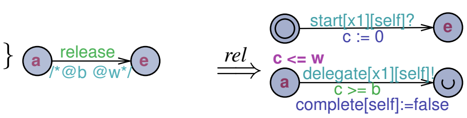
$[\![t!rec.m()\ /*@b\ @w\ @d*/]\!]_{a,e} = \{a \xrightarrow[c \geq b\ ;\ c:=0, deadline:=d]{invoke[t][m][rec][self]!} e\}, \{(a,w)\}$
where t is reused to represent the integer constant for label t.



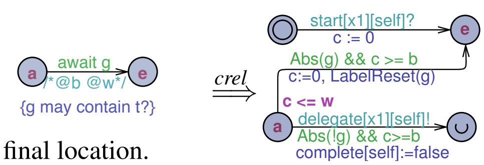
$[\![t?\ /*@b*/]\!]_{a,e} = \{a \xrightarrow[c \geq b]{wait[t][self]!} l_0,\ l_0 \xrightarrow[LabelReset(t?), c:=0]{resume[t][self]?} e\}, \{(a,b)\}$
where t is reused to represent the integer constant for label t.



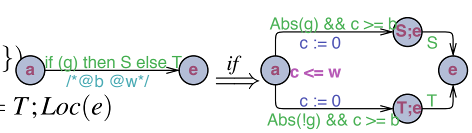
$[\![release\ /*@b\ @w*/]\!]_{a,e} =$
$\{a \xrightarrow[c \geq b\ ;\ complete[self]:=false]{delegate[x1][self]!} u,\ l_0 \xrightarrow[c:=0]{start[x1][self]?} e\}, \{(a,w)\}, \{(x1, true)\}$
where u is the unique final location and $x1$ is a new name.



$[\![await\ g\ /*@b\ @w*/]\!]_{a,e} =$
$\{\ a \xrightarrow[c \geq b\ \wedge\ Abs(!g)\ ;\ complete[self]:=false]{delegate[x1][self]!} u, l_0 \xrightarrow[c:=0]{start[x1][self]?} e,$
$a \xrightarrow[c \geq b\ \wedge\ Abs(g)\ ;\ c:=0, LabelReset(g)]{} e\}, \{(a,w)\}, \{(x1, Abs(g))\}$
where $x1$ is a new name, l_0 is the initial location and u is the unique final location.



$[\![if\ (g)\ then\ S\ else\ T\ \ /*@b\ @w*/]\!]_{a,e} = [\![S]\!]_{l_1,e} \uplus [\![T]\!]_{l_2,e} \uplus$
$(\{a \xrightarrow[c \geq b\ \wedge\ Abs(g)\ ;\ c:=0]{} l_1, a \xrightarrow[c \geq b\ \wedge\ Abs(!g)\ ;\ c:=0]{} l_2\}, \{l_1, l_2\}, \{(a,w)\}, \{\})$
New locations: l_1 and l_2 such that $Loc(l_1) = S;Loc(e)$ and $Loc(l_2) = T;Loc(e)$



$[\![while\ (g)\ do\ S\ od\ /*@b\ @w*/]\!]_{a,e} = [\![S]\!]_{l,a} \uplus$
$(\{a \xrightarrow[c \geq b\ \wedge\ Abs(g)\ ;\ c:=0]{} l, a \xrightarrow[c \geq b\ \wedge\ Abs(!g)\ ;\ c:=0]{} e\}, \{l\}, \{(a,w)\}, \{\})$
New location: l such that $Loc(l) = S;Loc(e)$

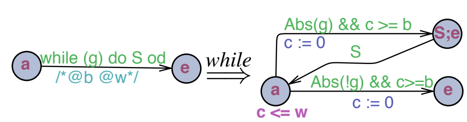


Figure 5: The rules for translating methods to timed automata. For each rule, we write only non-empty sets. The union of two tuples is defined as $(T,L,I,E) \uplus (T',L',I',E') = (T \cup T', L \cup L', I \cup I', E \cup E')$

$$
\begin{array}{lcl}
\mathit{starts} & = & \{E(m) \mid m \in m^*\} \\
\mathit{tasks} & = & \{N \mid \textbf{op}\ N == S \in m^*\} \cup \{n \mid (n, en) \in E(m), m \in m^*\}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{labels} = \bigcup \mathit{labels}(S) \text{ for } \textbf{op}\ N == S \in m^* & \mathit{methods} = \bigcup \mathit{mthds}(S) \text{ for } \textbf{op}\ N == S \in m^* \\
\mathit{labels}(s;S) = \mathit{labels}(s) \cup \mathit{labels}(S) & \mathit{mthds}(s;S) = \mathit{mthds}(s) \cup \mathit{mthds}(S) \\
\mathit{labels}(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2) = \mathit{labels}(S_1) \cup \mathit{labels}(S_2) & \mathit{mthds}(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2) = \mathit{mthds}(S_1) \cup \mathit{mthds}(S_2) \\
\mathit{labels}(\textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}) = \mathit{labels}(S) & \mathit{mthds}(\textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}) = \mathit{mthds}(S) \\
\mathit{labels}(t!p()\ /{*}@d{*}/) = \{t\} & \mathit{mthds}(t!x.m) = \{m\} \\
\mathit{labels}(\_) = \emptyset & \mathit{mthds}(\_) = \emptyset
\end{array}
$$

Figure 6: Encoding helper functions. As parameter _ represents other possibilities.

ret. The statement `t?` stops the current method and triggers the scheduler (on `wait` channel). If `t` is associated to a local call, the scheduler immediately starts the called method (i.e., sync call). This can be a recursive call because the *ret* rule moves the method to the initial location l_0 (transition from 16 to l_0 in Figure 3). If `t` refers to a remote call, the object is blocked (no method is executed). The caller is *resumed* when the callee has finished and a reply signal is available (transition from l_0 to 17 in Figure 3). Since the upper bound on this statement depends on the called methods, we can only express a lower bound (/*@b*/) on its execution time.

crel. With the `await g` statement, if `g` holds, the processor is not released and the method continues (e.g., transition 25 to 26 in Figure 3). If `g` does not hold, control moves to the final location u (e.g., from location 25); therefore, the current task will finish. Instead, it generates a subtask `x1` using the `delegate` channel (thus it inherits the deadline), with enabling condition `g`. The transition `start[x1][self]?` defines the subtask `x1` such that it will execute the rest of the original task (e.g., from l_0 to 26).

Global Declarations In Figure 6, we define some helper functions we need for computing the global declarations *Dec* for '**class** $C(a^*)$ **implements** i^* **begin** v^* m^* **end**'. To save space, we do not write the real-time execution information when not relevant for these function definitions. We use $E(m)$ as a short hand to return the element E returned by $[\![m]\!]$ for $m \in m^*$.

Given a minimum d_n, a maximum d_x and an initial value d_i for deadline, the global declarations *Dec* are defined as follows. The global declarations *Dec* consists of the following elements. For easier reading, we put them in a list. Treating each item as a set, *Dec* is formally defined as their union. The functions *IntT*, *IntL* and *IntM* below produce unique integer values, starting from zero (*IntL* starts from 1) and incrementing by one every time they are called.

- global constants. The # function returns the number of elements in a set.
  `const int MSG =` $\mathit{Max}(\#(\mathit{methods} \setminus \mathit{tasks}), \#(\mathit{tasks}))$`;`
  `const int nObj =` $\#(a^*)+1$`;`
  `const int LBL =` $\#(\mathit{labels})$`;`
- the global clock used by all method automata and the deadline variable.
  `clock c; meta int[`d_n, d_x`] deadline;`
- a unique number for each task and subtask.
  `{const int` t `=` $\mathit{IntT}()$`;` $\mid t \in \mathit{tasks}\}$
- a unique number for each label, and a boolean array.
  `{const int` l `=` $\mathit{IntL}()$`;` $\mid l \in \mathit{labels}\}$
  `bool labels[LBL+1][nObj];`

- an array for each variable, either a bool or an int.
  $\{type\ name$ `[nObj];` $\mid name : type \in v^*\ \}$
- a bool to indicate method completion, helping scheduler decide whether to issue a reply signal.
  `bool complete[nObj];`
- a unique number for each method called on the objects provided as arguments.
  $\{$`const int` m `=` $IntM()$`;` $\mid m \in methods \setminus tasks\ \}$
- the channels used by the scheduler.

```
chan delegate [MSG+1][nObj];
chan invoke [LBL+1][MSG+1][nObj][nObj];
urgent chan start [MSG+1][nObj];
chan finish[nObj];
chan wait[LBL+1][nObj];
urgent chan resume[LBL+1][nObj];
chan reply[LBL+1][nObj];
```

- code to help the scheduler start the tasks correctly.

```
bool isEnabled (int msg, int self) {
 {if (msg == n) return en;    | for all (n,en) ∈ starts }
  return true;
}
```

5 Analysis of Real-Time Objects

The generated timed automata fit our automata-theoretic framework for modular schedulability analysis of asynchronous objects [8, 9]. The extensions to the original framework are as follows. Methods (and their corresponding messages) have enabling conditions. The completion of each method is reported back to the caller with a reply signal; this enables modeling Creol synchronization mechanisms. Static labels are used to match replies with their originating calls. A blocking synchronization statement waiting for reply from a local call leads to a deadlock; instead, as in basic Creol, we transform this situation to the synchronous execution of the called method. On the generated timed automata, one can perform normal UPPAAL analyses like reachability; using the *Loc* function (see Table 1), automata locations can be traced back to the original Creol code. As the original framework is intended, one can perform schedulability analysis to check whether called methods can finish within the required deadlines.

Schedulability Analysis An object is an instance of a class together with a scheduler automaton. An object is schedulable, i.e., all tasks finish within their deadlines, if and only if the scheduler cannot reach the Error location with a queue length of $\lceil d_{max}/b_{min} \rceil$, where d_{max} is the longest deadline for any method called on any transition of the automata and b_{min} is the shortest termination time of any of the method automata [8]. However, schedulable objects usually need a much smaller queue length in practice.

We can analyze a closed system of multiple objects, but it may lead to state-space explosion. We can avoid that by analyzing one object in isolation; the modular analysis is explained in detail in [8]. In this case, we need to restrict the possible ways in which the methods of this object could be called. Therefore, we only consider the incoming method calls specified in its behavioral interfaces. Receiving a message from another object (i.e., an input action in the behavioral interface) creates a new task (for handling that message) and adds it to the queue. The behavioral interface does not capture (internal tasks triggered by) self calls. To analyze the schedulability of an object, one needs to consider both the internal tasks and the tasks triggered by the (behavioral interface, which abstractly models the acceptable) environment.

If a class implements multiple interfaces, we check schedulability with all interfaces together. Intuitively, that is because such a class should be able to take part in the protocols provided by these behavioral interfaces together. This is also the case in the coordinator example. We can generate the possible behaviors of an object by making a network of timed automata consisting of its method automata, behavioral interface automaton *B* and a concrete scheduler automaton.

Once an object is verified to be schedulable with respect to its behavioral interface, it can be used as an off-the-shelf component. To ensure the schedulability of a system composed of individually schedulable objects, we need to make sure their real use is *compatible* with their expected use specified in the behavioral interfaces. For the details of checking compatibility, we refer to our previous work [8, 9].

Example To be able to perform analysis on an object in isolation, we need the behavioral interface specifications; we consider the specification in Figure 2. The behavioral interfaces, the methods and a scheduler automaton are put together in UPPAAL. We verified this object for schedulability with an 'earliest deadline first' strategy. As a result, we found out that there must be at least an inter-arrival of SPEED=25 time units (SPEED in Figure 2); with this inter-arrival time, the methods need a deadline of at least MD=21 until synchronization is successful. Methods run and body meet their given deadlines of 10 and 50. In this case, we observe that no more than 7 queue slots is needed. It is interesting that a jitter for each of the messages causes nonschedulability, because in the long run, this message will be either too early or too late at some point for synchronization to take place in time.

We can furthermore model check the correctness of the algorithm. First of all, we can check whether m1 can go through its both release points; this is done by checking the reachability of location marked 26 in m1 automaton. Furthermore, we added counters for each method, counting the number of times it has synchronized. We can thus check whether, for instance, m1 can be in its third round, while m2 has performed only one round. To ensure that m1 is triggered fast enough, we add three instances of m1 at time zero. We observe that such a scenario is indeed not possible.

6 Conclusions and Future Work

We bridge the gap between high-level declarative automata theory and object-orientation; we elevate scheduling that is normally deeply buried in deployment infrastructure up in high-level modeling. Creol is a full-fledged object oriented modeling language strong in formal modeling but the great amount of nondeterminism makes model checking impractical. In this paper, we have extended Creol with real-time and the possibility of specifying scheduling strategies as a complementary feature; nondeterminism is reduced, among others, by specifying scheduling strategies. We presented a timed-automata semantics, which can be used for verifying real-time properties and in particular schedulability and timed reachability. This semantics supports features in Creol like processor release points, replies and synchronous method calls, multiple interfaces and loops.

To allow verification we only allow finite state models. We can put a bound on the length of the queues for schedulable systems; therefore using a finite queue length is not an issue. However, abstracting possibly infinite variables to finite ones may lead to transitions that would not be possible in the original rewrite semantics of Creol. This can only add possible behaviors, with respect to the rewrite semantics, therefore if we say a Creol program meets its deadlines we can be sure that it does.

The automata model of a class consists of the following: one automaton for every method defined in the class, behavioral interfaces describing the overall input/output behavior and a scheduler automaton with the desired scheduling strategy. Using automata, behavioral interfaces can specify non-uniformly recurring tasks rather than for instance periodic tasks or using pessimistic approximations [7].

As further work, we are looking into extending the original rewrite semantics of Creol with real-time for simulation purposes. Instead of automatically generating automata from Creol, one may manually create abstract automata models corresponding to the Creol models, like [5]; to ensure schedulability of the corresponding Creol model, we will study conformance between Creol and timed automata.

References

[1] Bernhard K. Aichernig, Andreas Griesmayer, Einar Broch Johnsen, Rudolf Schlatte & Andries Stam (2008): *Conformance Testing of Distributed Concurrent Systems with Executable Designs*. In: Formal Methods for Components and Objects (FMCO'08), LNCS 5751, Springer, pp. 61–81.

[2] Rajeev Alur & David L. Dill (1994): *A Theory of Timed Automata*. Theoretical Computer Science 126(2), pp. 183–235.

[3] Sanjoy Baruah & Theodore Baker (2008): *Schedulability analysis of global EDF*. Real-Time Syst. 38(3), pp. 223–235.

[4] Frank de Boer, Tom Chothia & Mohammad Mahdi Jaghoori (2009): *Modular Schedulability Analysis of Concurrent Objects in Creol*. In: Proc. Fundamentals of Software Engineering (FSEN'09), 5961, pp. 212–227.

[5] Frank S. de Boer, Immo Grabe, Mohammad Mahdi Jaghoori, Andries Stam & Wang Yi (2009): *Modeling and Analysis of Thread-Pools in an Industrial Communication Platform*. In: Proc. 11th International Conference on Formal Engineering Methods (ICFEM'09), LNCS 5885, Springer, pp. 367–386.

[6] Etienne Closse, Michel Poize, Jacques Pulou, Joseph Sifakis, Patrick Venter, Daniel Weil & Sergio Yovine (2001): *TAXYS: A Tool for the Development and Verification of Real-Time Embedded Systems*. In: Proc. CAV'01, LNCS 2102, Springer, pp. 391–395.

[7] Elena Fersman, Pavel Krcal, Paul Pettersson & Wang Yi (2007): *Task automata: Schedulability, decidability and undecidability*. Information and Computation 205(8), pp. 1149–1172.

[8] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia & Marjan Sirjani (2009): *Schedulability of asynchronous real-time concurrent objects*. J. Logic and Alg. Prog. 78(5), pp. 402 – 416.

[9] Mohammad Mahdi Jaghoori, Delphine Longuet, Frank S. de Boer & Tom Chothia (2008): *Schedulability and Compatibility of real time asynchronous objects*. In: Proc. RTSS'08, IEEE CS, pp. 70–79.

[10] Einar Broch Johnsen & Olaf Owe (2007): *An Asynchronous Communication Model for Distributed Concurrent Objects*. Software and Systems Modeling 6(1), pp. 35–58.

[11] Einar Broch Johnsen, Olaf Owe & Ingrid Chieh Yu (2006): *Creol: A type-safe object-oriented model for distributed concurrent systems*. Theoretical Computer Science 365(1-2), pp. 23–66.

[12] Christos Kloukinas & Sergio Yovine (2003): *Synthesis of Safe, QoS Extendible, Application Specific Schedulers for Heterogeneous Real-Time Systems*. In: Proc. ECRTS'03, IEEE CS, pp. 287–294.

[13] Orna Kupferman, Moshe Y. Vardi & Pierre Wolper (2001): *Module Checking*. Information and Computation 164(2), pp. 322–344.

[14] Marcel Kyas & Einar Broch Johnsen (2009): *A Real-Time Extension of Creol for Modelling Biomedical Sensors*. In: Proc. FMCO'08, LNCS 5751, Springer, pp. 42–60.

[15] Kim Guldstrand Larsen, Paul Pettersson & Wang Yi (1997): *UPPAAL in a Nutshell*. STTT 1(1-2), pp. 134–152.

[16] Bertrand Meyer (1992): *Eiffel: The language*. Prentice-Hall.

[17] Barbara Paech & Bernhard Rumpe (1994): *A new Concept of Refinement used for Behaviour Modelling with Automata*. In: Proc. FME '94, Springer-Verlag, pp. 154–174.

[18] Ingrid Chieh Yu, Einar Broch Johnsen & Olaf Owe (2006): *Type-Safe Runtime Class Upgrades in Creol*. In: Proc. FMOODS'06, LNCS 4037, Springer-Verlag, pp. 202–217.